Knit: Component Composition for Systems Software

Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, Eric Eide

University of Utah

Knit is a new component definition and linking language targeted at low-level systems code.

Components == pieces of code that I can plug together in various ways.

Be sure to say what component kits are not:

"Components are unlike libraries in that I can plug them together in various ways instead of just one fixed way."

Why Components?

- Everyone is writing too much code
  - Not enough code reuse
  - Hard to reconfigure
  - Hard to understand
  - Hard to test/verify

- Exceptions: Click, Scout, Ensemble, Fox, MMLite, OSKit, …

Why Not Components?

- Overhead
  - Runtime
  - Programmer time
- Advanced systems don't work with C
- Complex component interdependencies
  - Locking restrictions
  - Top/bottom-half
  - Bootstrap sequence

Suggest that most people in audience don't use components and guess that these are some of their reasons.

Point out that some of these problems seem to be more acute for systems code.

"But there's something different about systems code"

Simple approaches (COM one example) have problems

Advanced solutions solve this but..

And even they don't solve the problems we have with systems code

Goal of Knit Project

To make components practical for systems programming

… by breaking down these barriers to using components

Key to Achieving Goal

Static configuration language

- ◆ Enables error detection
- ◆ Enables optimization

Static – so we can analyze the code which makes it possible to detect cycles, detect configuration errors and optimize code.

Contrast with OO approaches (and Inferno's modules?) where system configuration is established at runtime – which restricts what you can do.

Weaker contrast with standard Unix build tools where most configuration decisions are, effectively, made when you write the code.

Be sure to say that dynamic reconfiguration and OO techniques are appropriate in some places – but you don't want to use it for everything because being that dynamic has such problems.

Target#1: The Utah OSKit [SOSP'97]

- Approximately 500 components:

  Device drivers, bootstrap code, TCP/IP stacks, filesystems, SNMP, etc.

- Doesn't impose architecture

- 10^6 lines of code from Linux, FreeBSD, NetBSD, Mach, Fluke, etc.

We've applied Knit to two component kits so far. The first and most important is our own OSKit. We've also applied Knit to 'Clack' which is a re-implementation of MIT's Click modular router.

Knit is intended to be a general purpose tool. But we considered it essential that it should work well for one particular application: The Flux OSKit (which we presented 3 years ago at SOSP).

A kit of parts for building OS kernels

Two important things to notice are:

1) The OSKit doesn't try to impose any particular view of the world on you. You can use it to build microkernels or monolithic kernels. You can build multi-threaded and single-threaded kernels. You can run interrupt handlers in their own thread or as plain ordinary interrupts. This is, of course, good but it's also a problem because it's hard to state hard and fast laws about how systems should be built.

2) We're dealing with a lot of code from a variety of sources so th ey're going to make a variety of assumptions about the environment they're used in and no-one is actually going to be an expert in using all the components. This is also a good thing (having lots of code) but, again, it's a problem (because you can't keep it all in your head at once).

These two properties make it hard to use the OSKit.

We could make it easier to build OSKit systems if we imposed a more restrictive framework.

Outline

- Introduction
- The Knit component model
  - ◆ Atomic units
  - ◆ Compound units
  - ◆ Automatic Initialization
  - ◆ Detecting Configuration Errors
- Implementation and Performance
- Open issues

Knit   7

Atomic Units [PLDI'98]

serve_cgi	**serve_file**

```
int serve_web(…) {
  if (…)
    serve_cgi(…);
  else
    serve_file(…);
}
```

- `-Ioskit`
- `-DKERNEL`
- `-DHAVE_CONFIG`

serve_web

Knit is an application of Flatt and Felleisen's Unit Model which was first applied to Scheme and Java.

The basic idea is simple enough: encapsulate a piece of code in a "unit" which explicitly lists the symbols imported into the unit (at the top) and exported from the unit (at the bottom).

Atomic units:

       Explicit imports/exports

       Refers to C, assembly and object files

       Contain compilation information

Be clear that units are extensions to scheme and Java – not part of standards
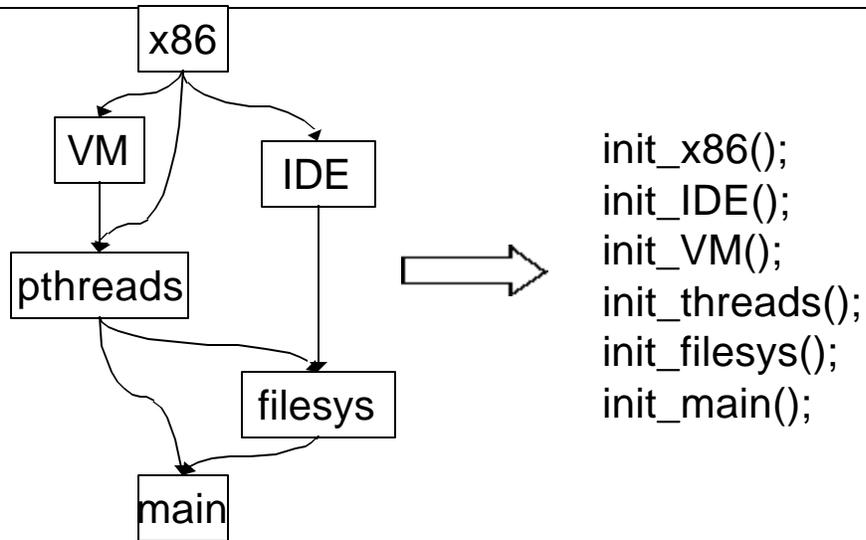
Compound Units [PLDI'98]

files　　　　　　**processes**

serve_file　　　serve_cgi

select_server

serve_web
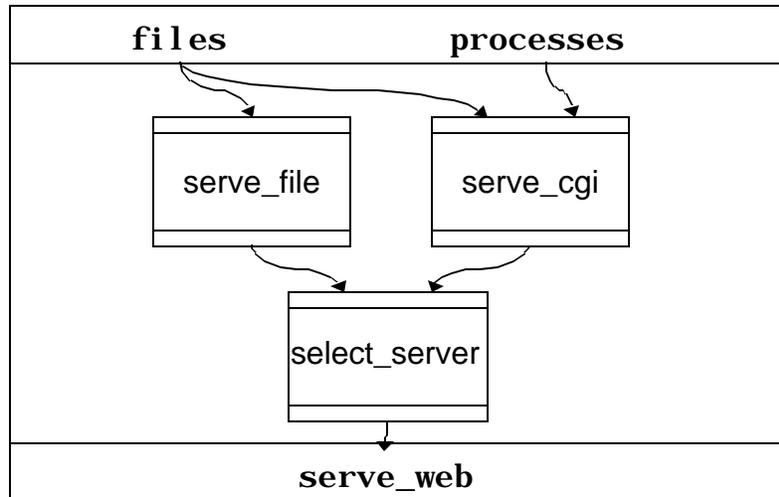
Compound units describe how we glue units together.

You can build hierarchies of components.

Initialization

x86

VM

IDE

pthreads

filesys

main

init_x86();
init_IDE();
init_VM();
init_threads();
init_filesys();
init_main();

So why did we have to do something about initialization?

Here's a vastly simplified picture of a kernel

Order depends on interconnections so every time you insert, remove or rearrange units, you have to change the initialization sequence.
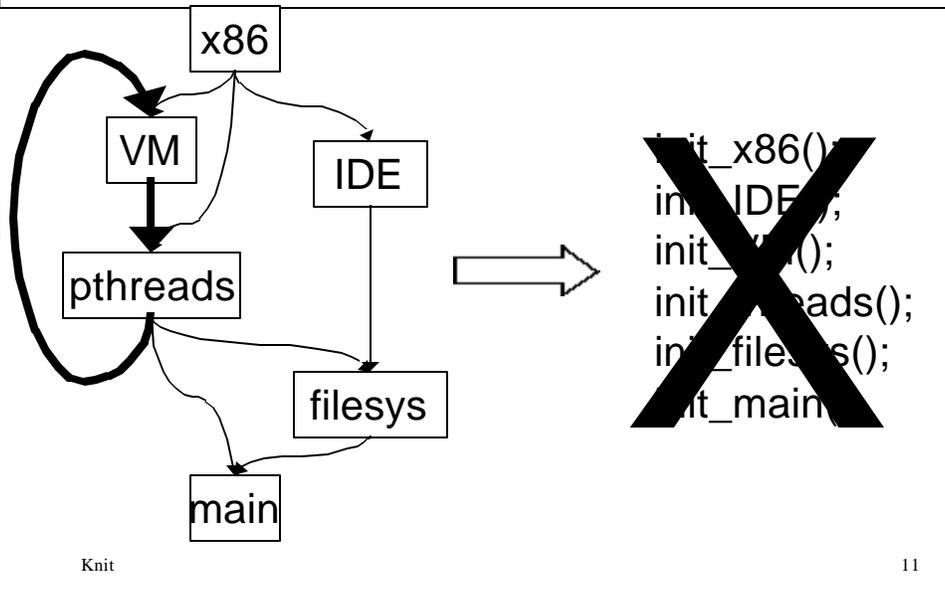
In practice, this was a tedious, error-prone bottleneck.

So Knit generates this code for you using the component interconnections to determine order.

In practice, OSKit programmers find this makes it much easier to reconfigure the system – encouraging experimentation.

Example is supposed to be complex enough that audience appreciates that automation is a good thing.
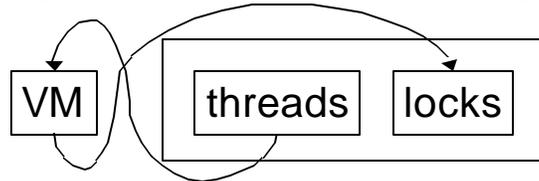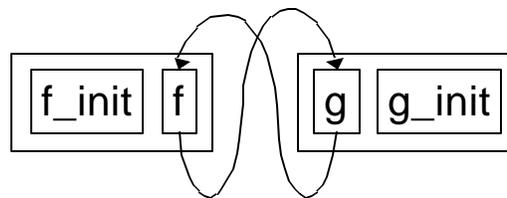
Initialization

x86

VM

IDE

pthreads

filesys

main

init_x86();
init_IDE();
init_VM();
init_threads();
init_filesys();
init_main();

The problem is what to do about cycles.

When Can We Break Cycles?

1. Component 'contains' subcomponents

   VM | threads | locks

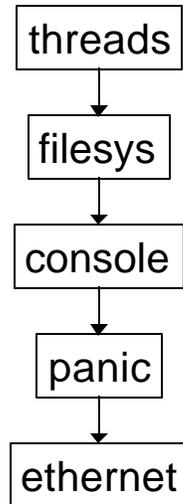2. No dependency between initializers

   f_init | f | g | g_init

Automatic Initialization

- Knit generates initialization sequence
- Cycles are resolved by refining initialization dependencies in units

- Experience
  - 5% of units need dependencies refined
  - Programmers find initialization a big win

Detecting Composition Errors

```
threads
   │
   ▼
filesys
   │
   ▼
console
   │
   ▼
panic
   │
   ▼
ethernet
```

Knit   14


Here's a small part of a kernel I built.

In this kernel, panic messages from our IDE device driver get logged to the filesystem.

I can see the experienced hackers all shaking their heads thinking "What an idiot! The interrupt handlers will log panics to the file too. Which means they'll have to take locks and, of course, interrupt handlers should not manipulate locks"

But I was young and foolish then and didn't know what I was doing.

And, besides, there were 100 other components in the system so I wasn't even thinking about device drivers.

Detecting Composition Errors

threads ← .... locks

↓

filesys

↓

console

↓

panic

↓

ethernet ← .... interrupts
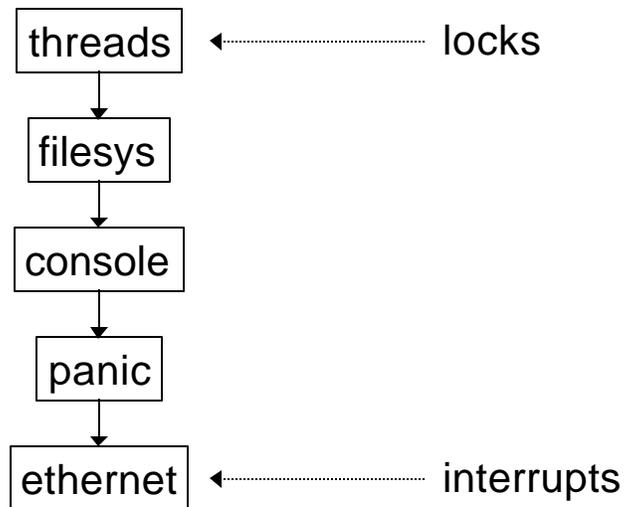
Knit   15

Use colors to indicate whether code is top half or bottom half (maybe not use those words)

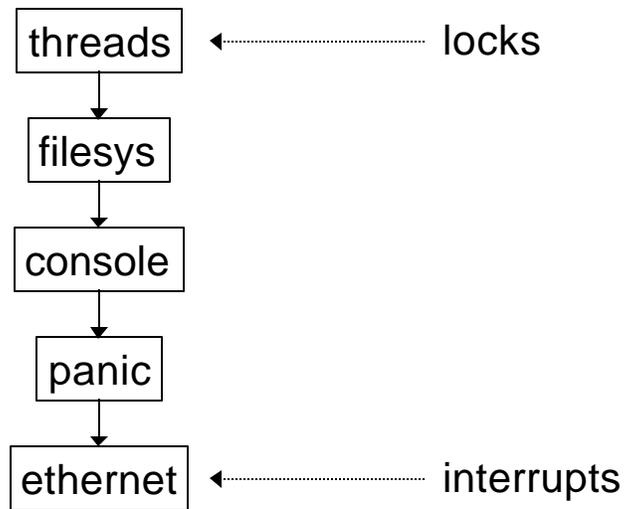Be sure to say that each connection is fine – it's the combination that is broken

Next few slides show how top-halfedness contaminates units which import it

Final slide shows a conflict: a unit which is both yellow and red.

Optional extra slide shows what we actually write

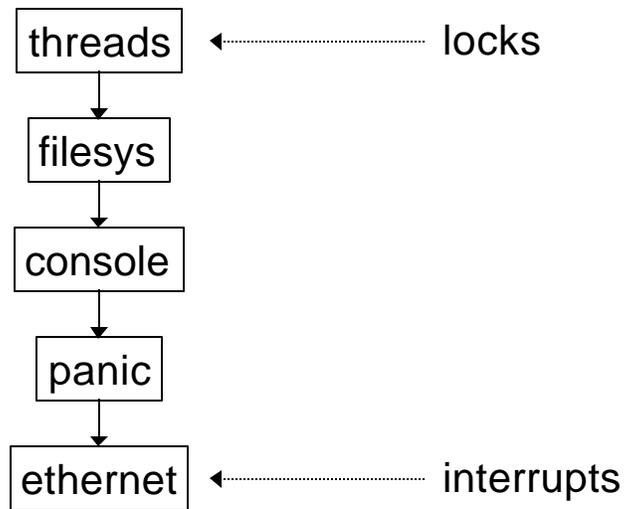Summary slide provides opportunity to abstract lessons out of this example
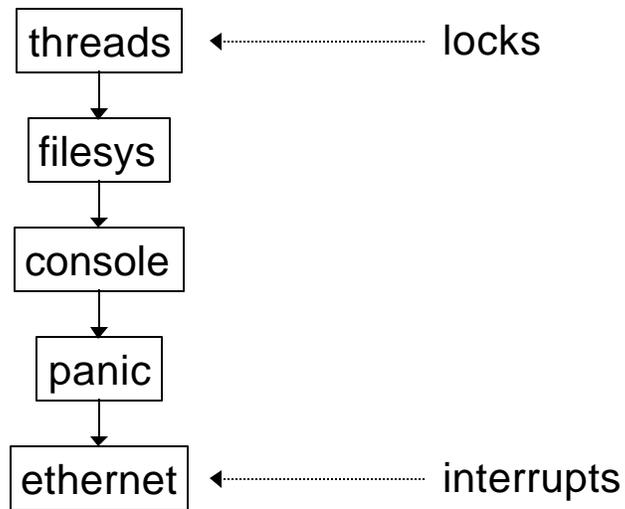
Detecting Composition Errors

threads ← ········· locks

filesys

console

panic

ethernet ← ········· interrupts

Detecting Composition Errors

threads ← ·········· locks

↓

filesys

↓

console

↓

panic

↓

ethernet ← ·········· interrupts

Detecting Composition Errors

threads ← - - - - - - - - locks

↓

filesys

↓

console

↓

panic

↓

ethernet ← - - - - - - - interrupts

Detecting Composition Errors

threads ← locks

↓

filesys

↓

console

↓

panic

↓

ethernet ← interrupts

Detecting Composition Errors

threads ◄········· locks

filesys

console

panic
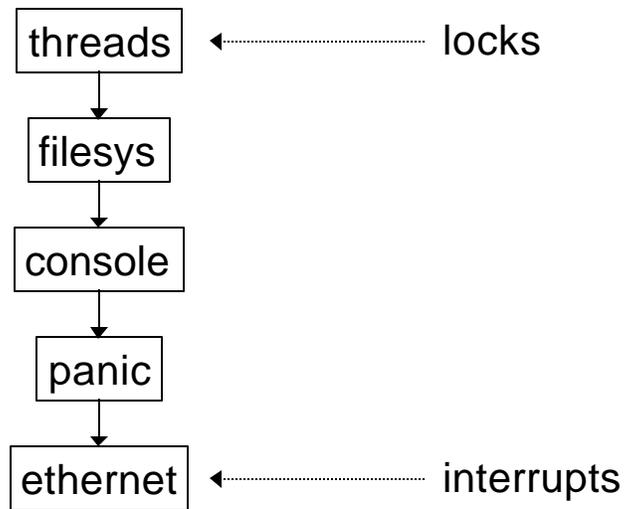
ethernet ◄········· interrupts
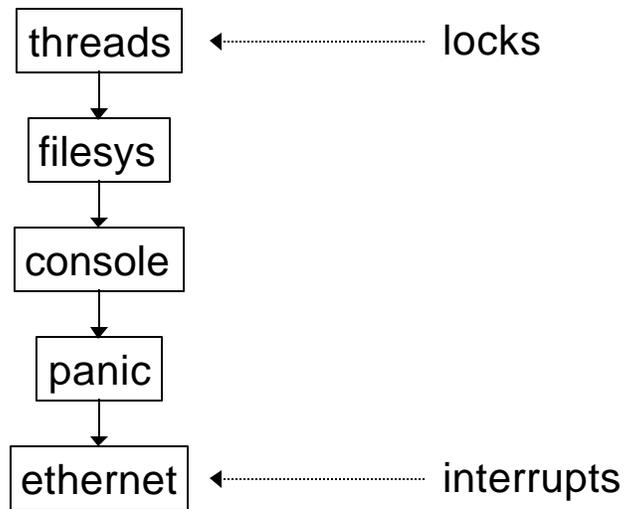
Knit 20

Detecting Composition Errors

threads   context(threads) <= ProcessContext

filesys   context(filesys) <= context(threads)

console   context(console) <= context(filesys)

panic   context(panic) <= context(console)

ethernet   NoContext <= context(ethernet)
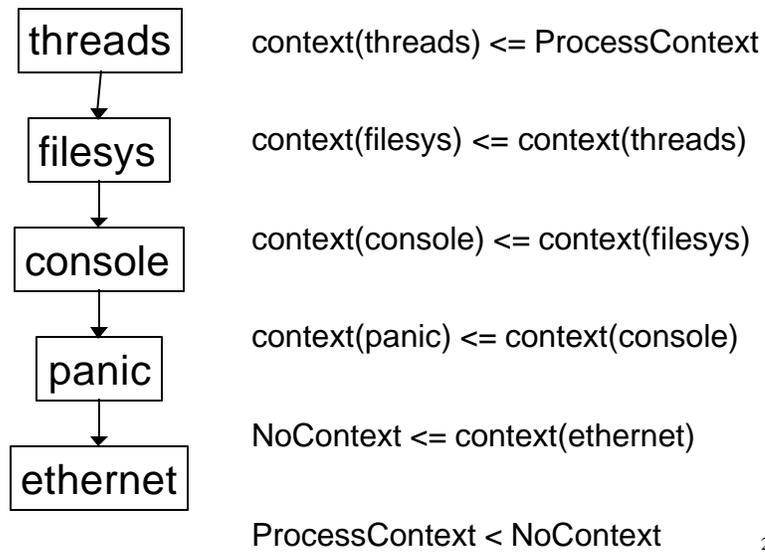
Knit   ProcessContext < NoContext   21

Extensible Constraint System

- Constraint system propagates properties through component interconnections
  - Knit can detect global errors
- Constraint system is extensible
  - In context X, don't do Y
  - Type system for Modular IP Routers (e.g., Click)
  - …

Knit   22

Could characterize as:

In context X, don't do Y:

1)top-half/bottom-half checks

2)restrictions on interrupt handlers

3)locking restrictions

Also used to encode a simple type system.

Knit

- Supports C, assembly and object files
- Separates interconnections from code
- Automatic initialization
- Extensible constraint system
- Allows cyclic component dependencies
- Allows multiple instances of components
- Text based

Knit   23

Last two are technical points that distinguish us from other approaches

Haven't motivated cycles and multiple instantiation yet

Should I add initialization, constraints and performance to this slide so that they see complete picture of what Knit does?


ToDo: quick comparison with advanced languages:

"Module systems in advanced languages give you almost all this but they don't address needs of systems software by supporting complex bootstrap sequences, constraints and paying close attention to performance."

Outline

- Introduction
- The Knit component model
- Implementation and Performance
- Open issues

Knit

Implementation (Unoptimized)
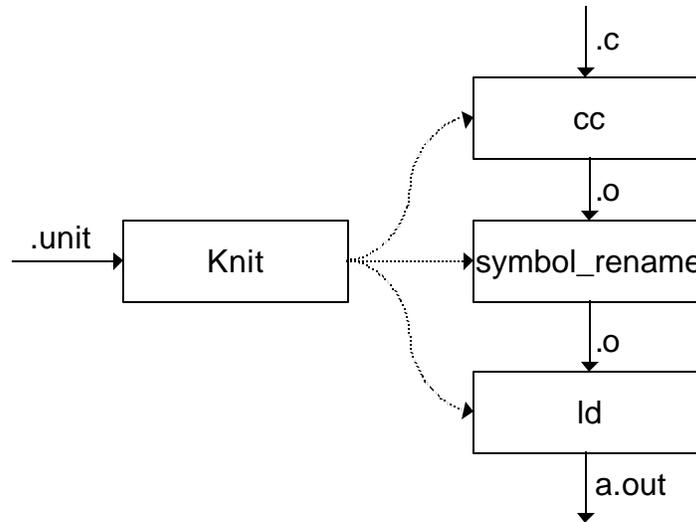
Knit   25

Knit read unit files.

Unit description directs compilation and linking process:

1) Uses standard C compiler to generate object files

2) Uses a tool we wrote to rename symbols in object files

3) Uses standard Unix linker to link program

Given this implementation, it's obvious that the cost of crossing component boundaries is just a function call.

So Knit achieves our goal of imposing a very low performance overhead.

Performance

- Component cost should not distort system structure
- Reduce overhead by eliminating function calls

The performance overhead of components affects the way they are used.

The more expensive it is to cross from one component to another, the larger your components are going to be.
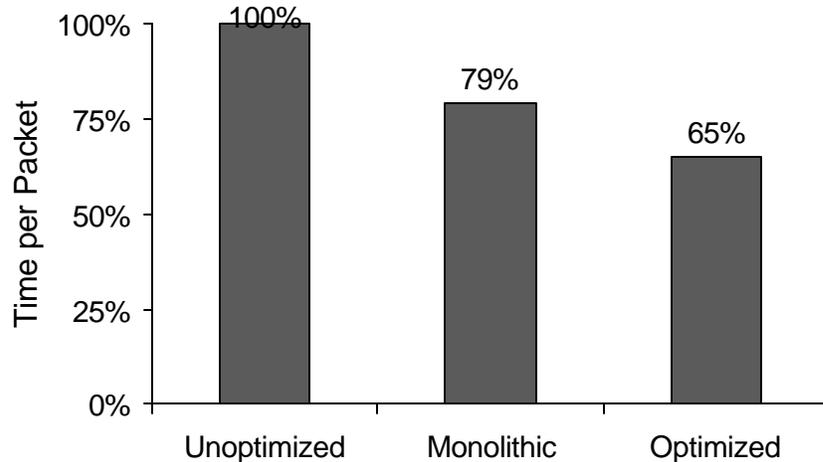
Click and Clack

- Click modular network router from MIT [SOSP'99]
- Clack
  - Re-implementation of Click using Knit
  - Similar performance to Click

- Many small components

Experiment to investigate performance overhead and show how inlining can help

Performance of Clack

100%

100%

79%

65%

Time per Packet

75%

50%

25%

0%

Unoptimized   Monolithic   Optimized

Knit   28

Clack is an IP router in the style of MIT's Click.

In fact, much of the design and code is lifted directly from the Click distribution.

For the purposes of this talk, the only thing you need to know about Clack is that it is made up of many small components. Some components are as trivial as setting or testing a field in a packet.

We measured the time it took for a packet to get from an input device to an output device.

(This path goes through about 13 or 14 components)

Discuss graph then…

Unoptimized == many small components, no cross-component inlining

Monolithic == manually combined the components into a small number of large components. Not recommended programming practice but it does go faster so performance is a problem and the temptation is there to let performance issues affect design.

Optimized == same small components as unoptimized case but using 'flattener' to perform cross-module inlining.

Point out that monolithic probably underestimates overhead. But point out that this is for small components and isn't what we see in the OSKit.

Open Issues

- Is Knit general purpose?
  - Need more users
  - Need more applications
- Is the constraint system extensible enough?
- Implicit linking vs. explicit linking?

Knit

So is Knit finished?

No, this is only the beginning.

Does Knit work for _your_ code?

Can you encode _your_ problems in the constraint system?

Knit insists that you say things like exactly which network stack each component is connected to – but since we only have one network stack in most systems, this gets old pretty fast.

Constraints work well in static configurations – there's various ways of handling more dynamic configurations with what we have at the moment – the interesting bit is how to make constraints more powerful _and_ handle dynamic configurations.

Conclusions

- State of the art component system for C
- Targeted at systems code
  - ◆ Automatic initialization
  - ◆ Detects local and global errors
  - ◆ Low performance overhead

- Available ASAP: http://www.cs.utah.edu/flux/

Knit

Btw We don't claim performance improvement because you could use same approach with plain C code without needing to use units at all. (But this wouldn't work for function pointer-based programming, C++, etc.)