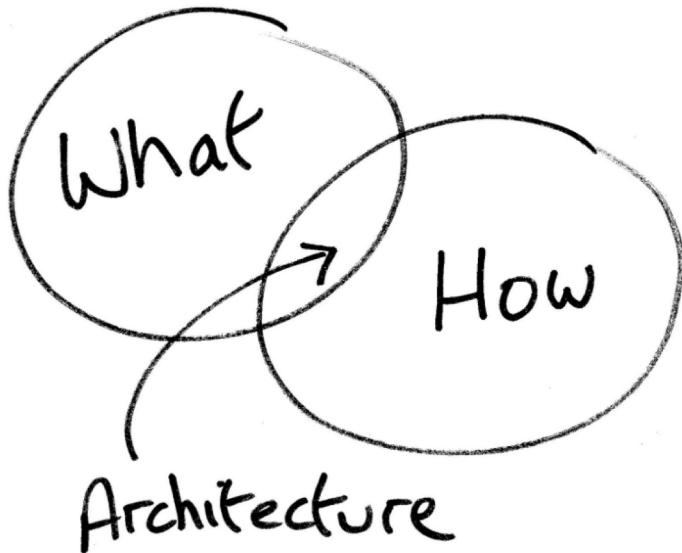




Software Architectures In Action



Software Architectures



- Coming up with good-quality designs and architectures is largely a matter of experience
- Best way to learn is to look at other people's work
 - ▶ Capture context and motivations
 - ▶ Understand the requirements
 - ▶ Follow the reasoning
 - ▶ Grasp the solution



The NginX Web Server (read “engine x”)



It's 1990...

- Web pages are small
 - ▶ A few kbytes of raw data
 - ▶ No multimedia content
 - ▶ Little to no server-side processing
- Internet connections are slow
 - ▶ Typically a few KBytes/s
- Client-server interactions are not persistent
 - ▶ Each query to a website is independent of past queries



Major performance bottleneck: **data transfer**



The Apache Web Server

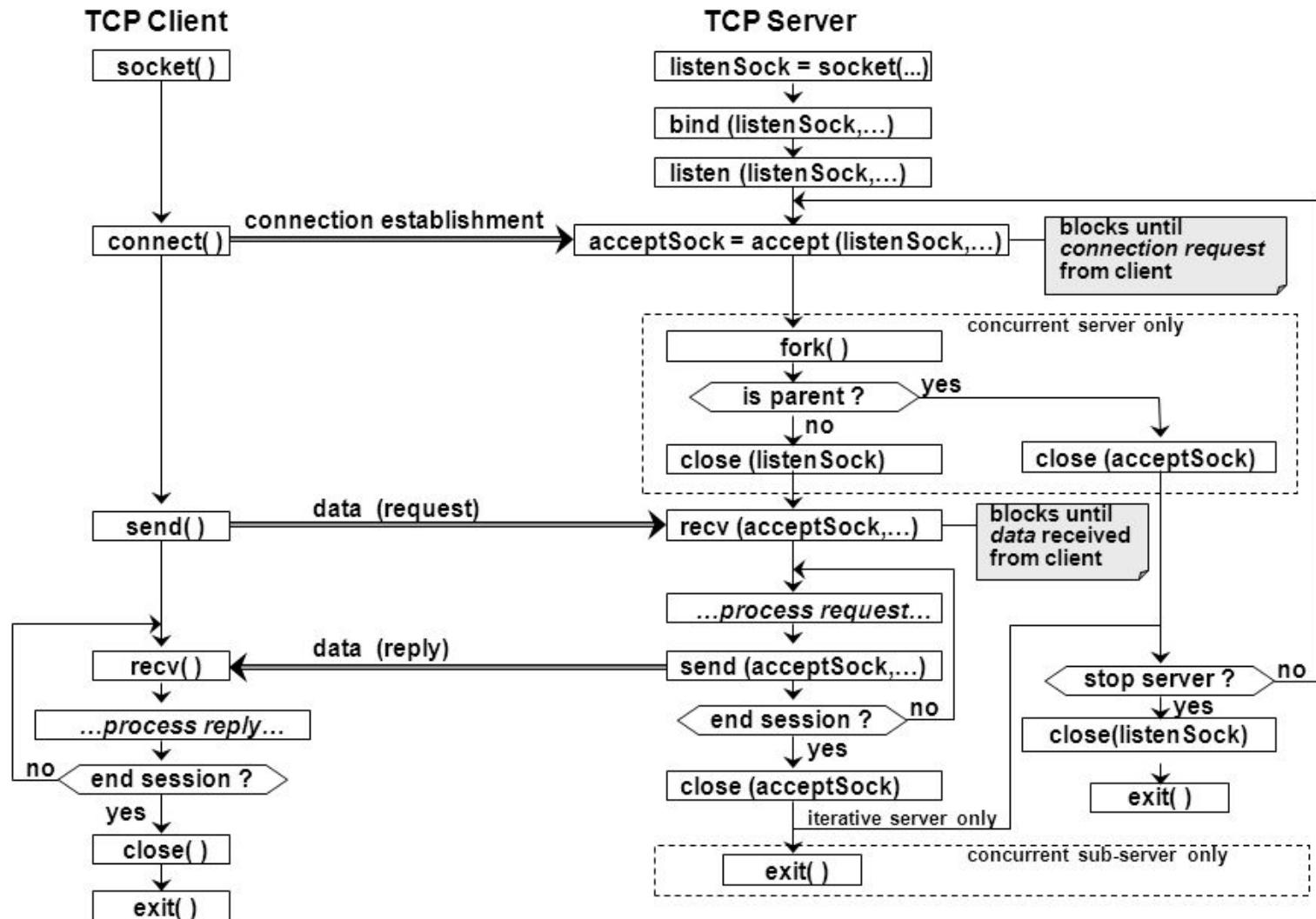
- Written using traditional Unix-style “process spawning” architecture
 - ▶ Every new connection from a client is handled by a separate concurrent process



APACHE



Apache Architecture (Simplified)





Apache: Pros and Cons

- Pro: easy to implement and test
 - ▶ Sequential code semantics
 - ▶ Blocking operations
- Pro: deterministic inter-process coordination
 - ▶ Wait()- and Notify()-style
 - ▶ If at all necessary...
- Cons: every process has its own process image
 - ▶ Copied from parent process
 - ▶ Global data, stack, heap, ...
- Cons: many more processes than cores



Time Passes...

- Connections improve by orders of magnitude
 - ▶ GBit/s subscription for **residential access** exist now
- Web pages become heavier
 - ▶ Full-fledged applications through the web browser
- Web applications become **stateful** and **push** data to the clients through **permanent** HTTP connections
- Modern web browsers parallelize queries to server to speed up rendering





Are Multi-proc Architectures Still Up?

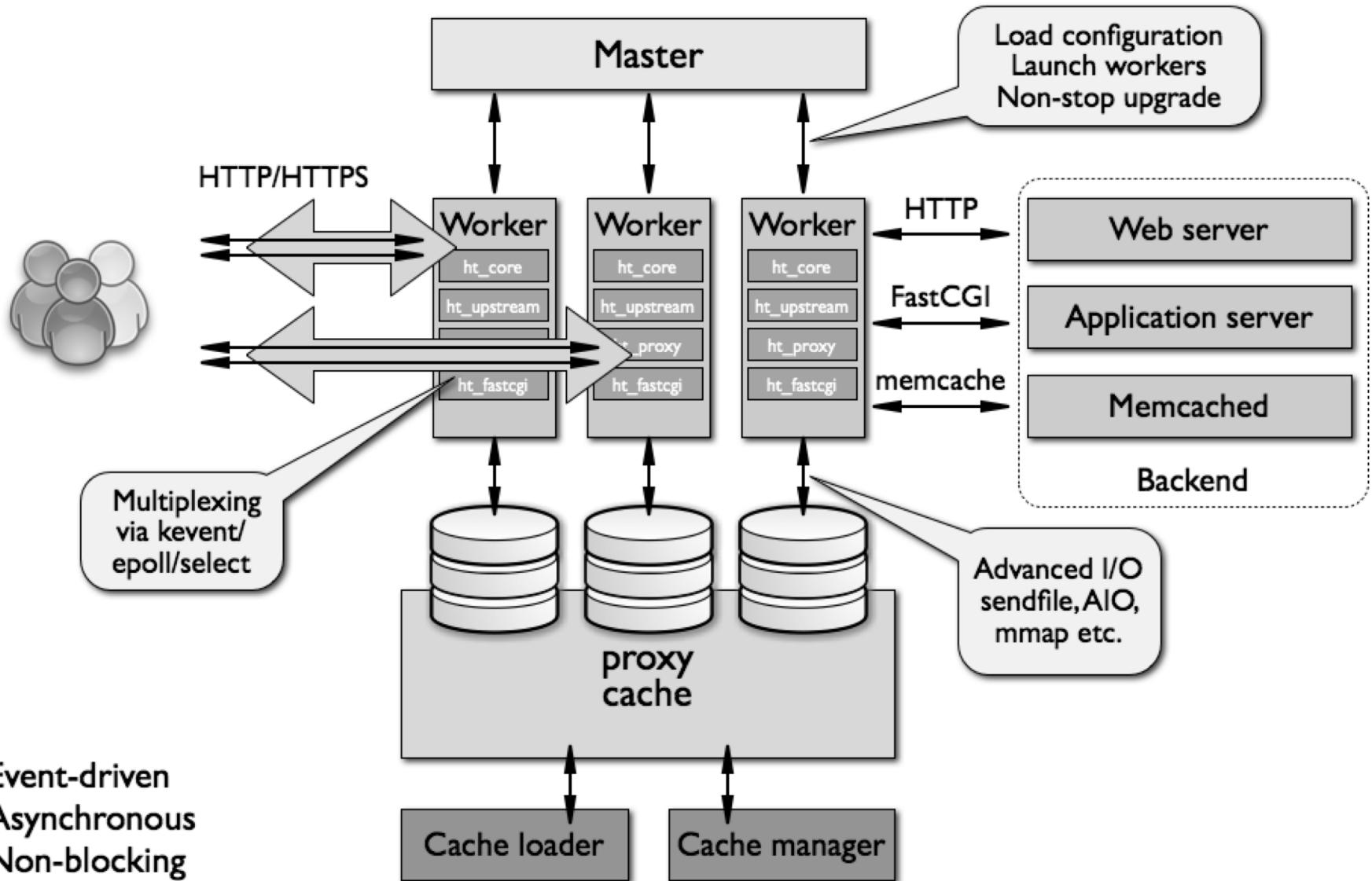
- **Tens of thousands** of connections need to be handled in parallel
- Connections are no longer the bottleneck in many cases
 - ▶ Even in mobile, think of 4G connections
- **Concurrency/parallelism** on the server side start to become challenges
- With plain Apache 2.2 (no optimizations), consider
 - ▶ ...a (optimistic) memory allocation of 2 MBytes per process
 - ▶ ...each process serving a 1 MByte page
 - ▶ ...Apache's core memory consumption (between 10 and 100 Mbytes)
 - ▶ ...typical memory consumption of stock Linux kernels, network stacks, and storage handling
- Serving 1K clients in parallel may require 128+GBytes of RAM



NginX

- No multi-processing architecture, but **event-driven!**
- Interesting events are, for example:
 - ▶ A new connection opening
 - ▶ An existing connection closing
 - ▶ An application server returning a result for a client
 - ▶ Storage operations commencing or concluding
- Number of processes are fixed and defined by the system's administrator
- Every process
 - ▶ ...waits for any interesting event
 - ▶ ...handles it quickly by using non-blocking operations as much as possible
 - ▶ ...returns to await other interesting events

NginX Architecture



- Event-driven
- Asynchronous
- Non-blocking

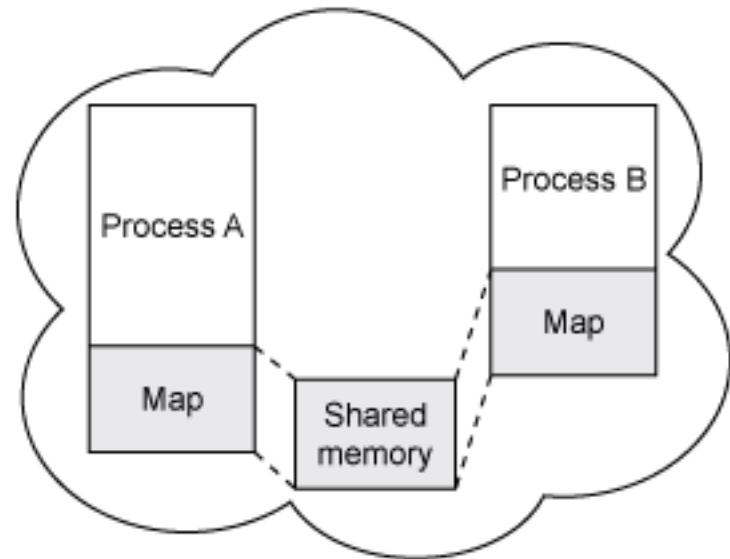


NginX Workers

- Aim at increasing parallelism by not locking the CPU
- One worker handles multiple clients
 - ▶ ...it can do so because while waiting for some operation to complete for a client, it can do something else for a different client
 - ▶ Example: while the disk loads a picture to send off to client X, the same worker can accept a new connection from a different client
- The approach works as long as the underlying OS offers **non-blocking** system calls and corresponding notifications to indicate completed operations
- Very conservative memory-wise
 - ▶ No create-destroy of process images as clients come and go
- One worker per core and the system achieves maximum CPU

Coordination Between Workers

- Based on **shared memory**
- Decouples the execution of processes, further increasing parallelism
- Can operate in a non-blocking fashion if a shared-memory manager notifies processes of changes in interesting portions





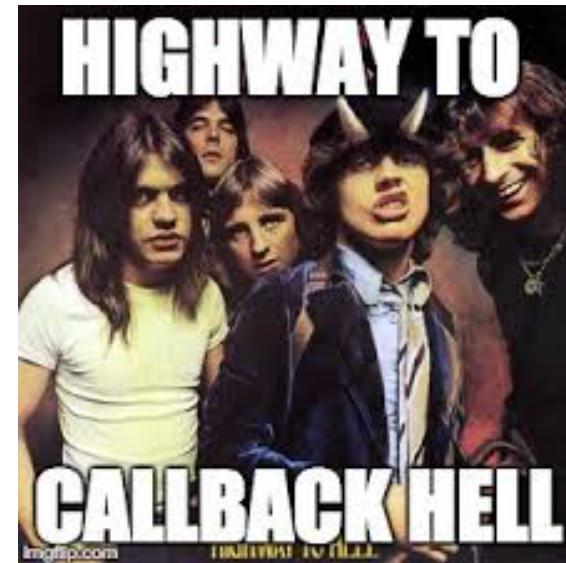
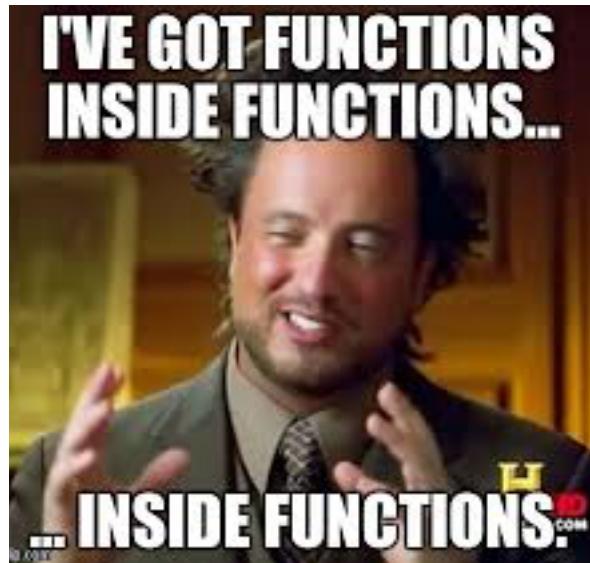
NginX: Pros and Cons

- Pro: very **conservative memory-wise**
 - ▶ No continuous create-destroy of process images as clients come and go
- Pro: one worker per core and the system achieves **near-optimal CPU utilization**
 - ▶ Dimensioning is easier
 - ▶ From the Ngnix documentation: “if the load is CPU intensive, the number of workers should match the number of CPU cores; if the load is disk I/O bound, the number of workers may be 1.5 times the number of CPU cores”
 - ▶ **Orders of magnitude improvements** in the number of manageable concurrent clients



NginX: Pros and Cons

- Cons: extremely **difficult to implement and test**
 - ▶ Event-driven processing loses the sequential semantics!





NginX: What's To Learn?

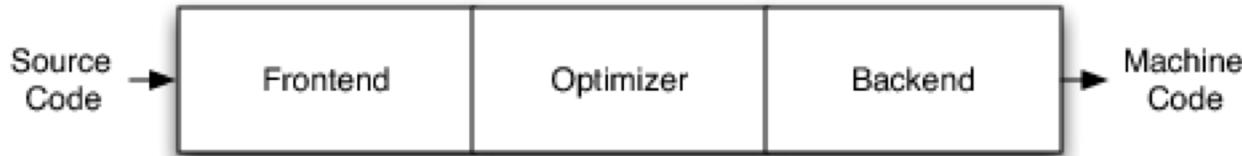
- Drastic performance improvements are enabled by a **better architecture**
 - ▶ Not everything boils down to code, rather the opposite...
- The architecture stays, the code goes...
 - ▶ The event-driven architecture of NginX remained essentially the same from the initial version more than 14 years ago
 - ▶ The code got almost entirely re-written as new OS APIs with variable semantics become available
- Btw, Apache is also partly transitioning to an event-driven architecture
 - ▶ Recall architecture drift and erosion
 - ▶ A case of “fixing a square peg in a round hole”...



LLVM



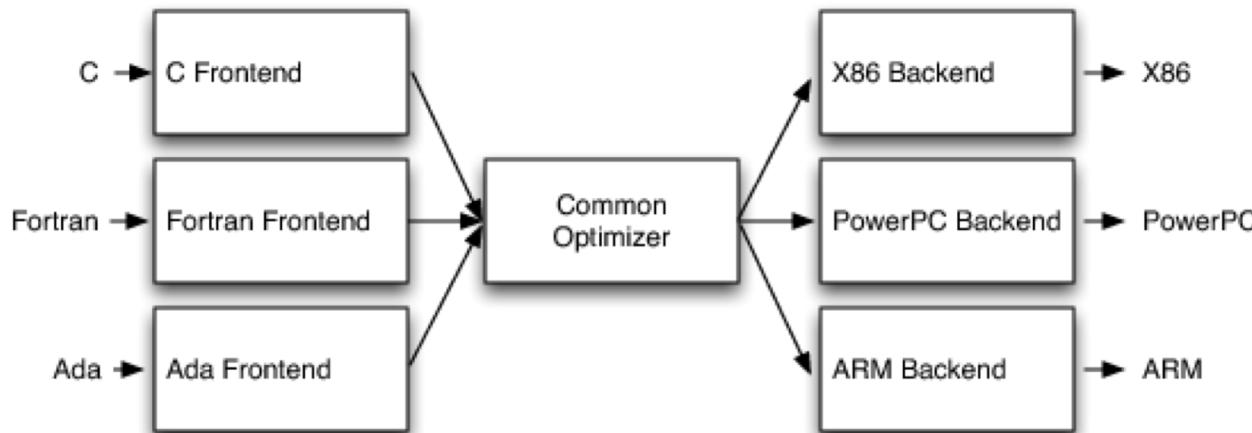
How a Compiler Should Look Like



- **Three-stage pipelined architecture**
 - ▶ Frontend: parses the code, checks for syntax errors, builds an abstract syntax-tree (AST)
 - ▶ Optimizer: takes the output of the frontend and improves it for efficiency, memory consumption...
 - ▶ Backend: generates machine code from the optimized output

Benefits of the 3-Stage Pipeline

- Applies to compilers producing binary code as well as Just-In-Time (JIT) compilers and interpreters
 - ▶ The **architectural abstraction** is the same for both C and Java, for example
- Fosters **separation of concerns**: the skills for implementing front-ends and back-ends are way different!
- Allows individual stages in the pipeline to be swapped in/out...
 - ▶ ... provided the stages remain **decoupled** and the input and output interfaces are **consistent**





The Sad Reality

- It is extremely difficult to decouple the three stages
- Very few existing compilers achieve a good level of decoupling
 - ▶ Java:
 - the back-end (JVM) is tightly coupled with the optimizer
 - the optimizer assumes a certain semantics of the JVM to apply semantics-preserving optimizations
 - ▶ GCC:
 - the back-end walks the front-end AST to produce debug info
 - the front-end produces data structures used by the back-end
 - global variables drive the entire process across the three stages
- These are cases known as “layering problems” and “leaky abstractions”



LLVM

- LLVM is the first (and most successful) effort so far to fully decouple the three stages
- **Key observation:** decoupling happens as long as the format of data flowing through the pipeline is sufficiently expressive not to force any of the stages to “poke” in any of the others
- Solution: design a **full-fledged** programming language that can be used **across the pipeline** as the only means to exchange information between subsequent stages





LLVM

- LLVM is the first (and most successful) effort so far to fully decouple the three stages
- **Key observation:** decoupling happens as long as the **encoding of data** flowing through the pipeline is sufficiently expressive not to force any of the stages to “poke” in any of the others
- Solution: design a **full-fledged** programming language that can be used **across the pipeline** as the only means to exchange information between subsequent stages





The LLVM Intermediate Language (IR)

An example of C code, two toy functions for adding two numbers

```
unsigned add1(unsigned a, unsigned b) {
    return a+b;
}

unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}
```



The LLVM Intermediate Language (IR)

The corresponding encoding in the IR

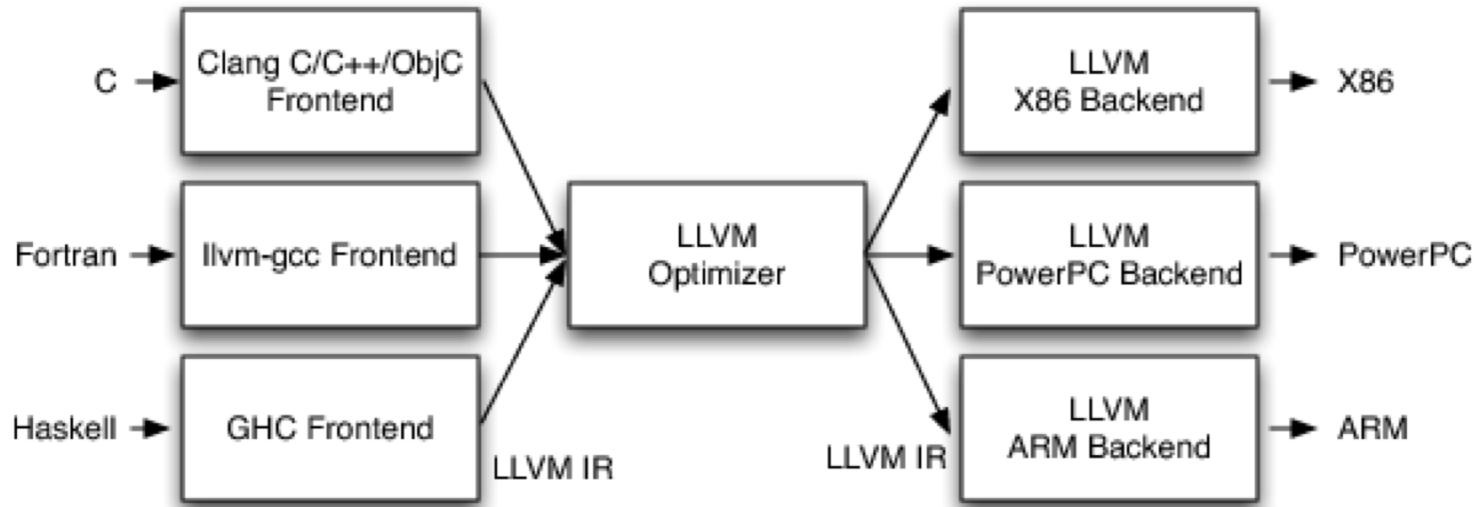
```
define i32 @add1(i32 %a, i32 %b) {
entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse

recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

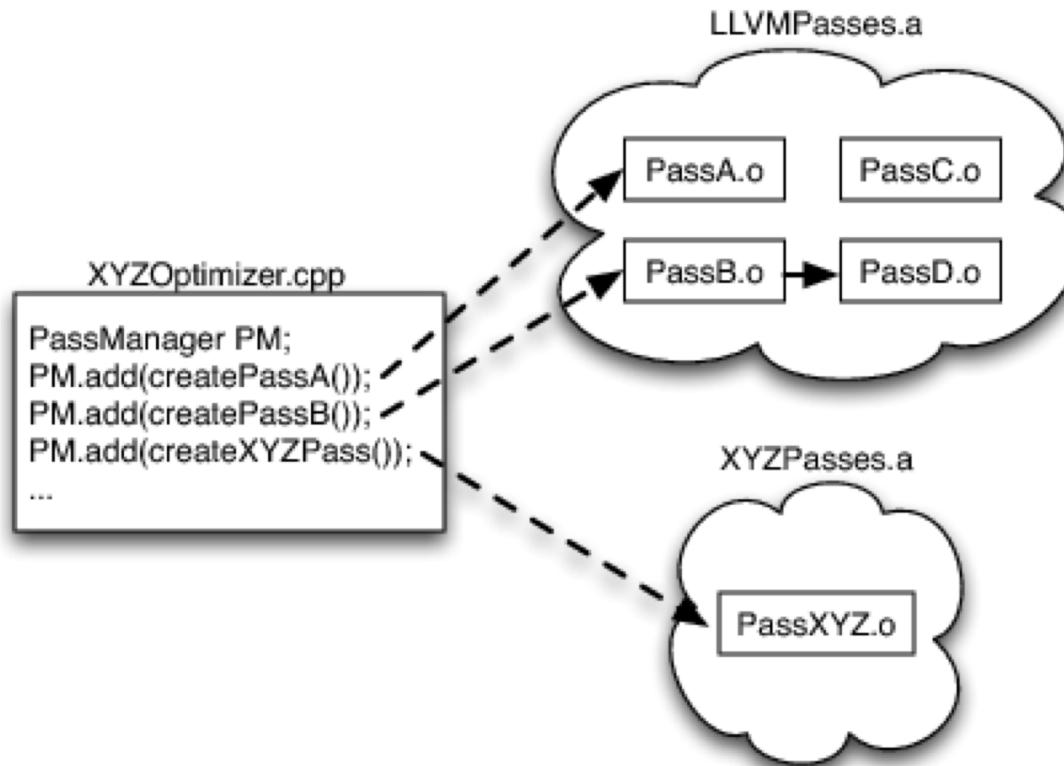
LLVM Architecture



- All that front-end developers need to know is LLVM IR
- All that back-end developers need to know is LLVM IR
- All that optimizer developers need to know is.. got it?

Configurability of the optimizer

- It is possible to apply various optimizations in different passes
- Users can implement new passes for specific optimizations



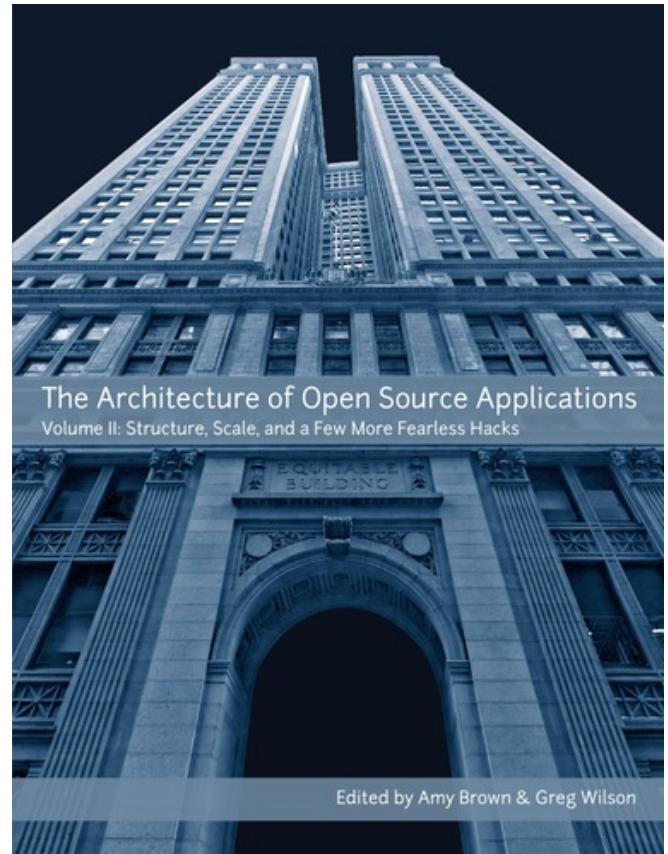


What's The Deal?

- Enables **unit-testing** of the individual stages
 - ▶ ...remember "design for testability"?
- LLVM IR allows for a pure **text-based** representation
 - ▶ Overcomes binary format and memory mapping problems
 - ▶ Allows for the decoupling of the execution of the three stages in time and space!
 - Run the front-end on a machine, save the result in a file, copy it over to two other machines, run different back-ends on these!
- Again, GCC also is developing along these lines, yet the intermediate representation (so-called GIMPLE rules) is not yet fully self-contained



More Examples



<http://www-aosabook.org/en/>